Procedia Computer Science

Volume 29, 2014, Pages 2168–2181

ICCS 2014. 14th International Conference on Computational Science

Multi-tenant Elastic Extension Tables Data Management

Haitham Yaish[a,b], Madhu Goyal[a,b], George Feuerlicht [b,c]

[a] Centre for Quantum Computation & Intelligent Systems

[b] Faculty of Engineering and Information Technology
University of Technology, Sydney
P.O. Box 123, Broadway NSW 2007, Australia

[c] Faculty of Information Technology,
University of Economics, Prague, Czech Republic
haitham.yaish@student.uts.edu.au, madhu@it.uts.edu.au,
george.feuerlicht@uts.edu.au

Abstract

Multi-tenant database is a new database solution which is significant for Software as a service (SaaS) and Big Data applications in the context of cloud computing paradigm. This multi-tenant database has significant design challenges to develop a solution that ensures a high level of data quality, accessibility, and manageability for the tenants using this database. In this paper, we propose a multi-tenant data management service called Elastic Extension Tables Schema Handler Service (EETSHS), which is based on a multi-tenant database schema called Elastic Extension Tables (EET). This data management service satisfies tenants' different business requirements, by creating, managing, organizing, and administrating large volumes of structured, semi-structured, and unstructured data. Furthermore, it combines traditional relational data with virtual relational data in a single database schema and allows tenants to manage this data by calling functions from this service. We present algorithms for frequently used functions of this service, and perform several experiments to measure the feasibility and effectiveness of managing multi-tenant data using these functions. We report experimental results of query execution times for managing tenants' virtual and traditional relational data showing that EET schema is a good candidate for the management of multi-tenant data for SaaS and Big Data applications.

Keywords: Cloud Computing, Software as a Service, Big Data, Elastic Extension Tables, Multi-tenant Database, Relational Tables, Virtual Relational Tables.

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.202

1 Introduction

Multi-tenancy is the fundamental characteristic of Software as a Service (SaaS), which allows SaaS vendors to run a single application supporting multiple tenants on the same hardware and software infrastructure [9, 13]. The application layer of SaaS has four maturity model levels: (1) Ad-Hoc/Custom, (2) Configurable, (3) Configurable and Multi-Tenant-Efficient, and (4) Scalable, Configurable, and Multi-Tenant-Efficient [1, 4, 13]. In this paper, we adopt the Configurable and Multi-Tenant-Efficient Level to design our data management service. Configuration is the main characteristic of multi-tenant applications that allows SaaS vendors running a single instance application, which provides the means of configuration for multi-tenant applications. This characteristic requires a multi-tenant aware design with a single codebase and metadata capability. Multi-tenant aware application allows each tenant to design different parts of the application, and automatically adjust and configure its behavior during the application runtime execution without redeploying the application [2]. However, not all SaaS vendors provide a configuration capability for multi-tenant SaaS applications, and it might be ad-hoc and manual configuration practices [10]. Multi-tenant data has two types: shared data and tenant's isolated data; combining these two types of data together, gives a complete data view for the tenants that fit their business requirements [3, 11].

In our earlier work we proposed a configurable database design technique for multi-tenant applications - Elastic Extension Tables (EET) that consist of Common Tenant Tables (CTT), Virtual Extension Tables (VET), and Extension Tables (ET) [5, 8]. Additionally, we proposed an architecture design to build a database middleware to be used between software applications and Relational Database Management Systems (RDBMS). Using this database middleware, tenants can store, access and manage their data in the EET [7]. The main contribution of this paper is proposing the data management service of this architecture that called Elastic Extension Tables Schema Handler Service (EETSHS). This service enables its tenants to build their own virtual database schema using a four-step procedure: (1) creating the required number of tables and columns, (2) creating virtual database relationships, (3) assigning suitable data types and constraints for table columns, and (4) managing CTT and VET rows during multi-tenant application run-time execution. Furthermore, we propose three algorithms to manage CTT and VET by inserting, updating, and deleting rows from these two types of tables. Finally, we perform three experiments to verify the practicability and the effectiveness of using EETSHS service and EET.

The remainder of the paper structured as follows. Section 2 describes Elastic Extension Tables multi-tenant database schema. Section 3 proposes our multi-tenant data management service. Section 4 gives our experimental results, and Section 5 concludes this paper and describes future work.

2 Elastic Extension Tables

The proposed Elastic Extension Tables (EET) database schema is a novel way of designing and creating a multi-tenant database, which consists of three types of tables [5, 8]. The first type are Common Tenant Tables (CTT) which are physical tables shared between tenants using RDBMS. These physical relational tables can be applied to any business domain database such as Customer Relationship Management (CRM), Accounting, Human Resource (HR), or other business domains. The second type are Virtual Extension Tables (VET) that allow tenants to extend the existing business domain database, or have their own configurable database through creating their virtual database structures from scratch by creating (1) virtual database tables, (2) virtual database relationships, and (3) other database constraints. The third type are Extension Tables (ET) that consist of eight physical tables that are used to construct VETs [5, 8].

The design details of the eight extension tables of EET are shown in Figure 1 and listed as follows: (1) the 'db_table' extension table allows tenants to create virtual tables and give them unique names.

(2) The 'table_column' extension table allows tenants to create virtual columns for a virtual table stored in the 'db_table' extension table. (3) The table row extension tables store records of virtual extension columns in three separate tables. These tables are separated to store small data values in the 'table_row' extension table such as NUMBER, DATE-and-TIME, BOOLEAN, VARCHAR and other data types. The large data values stored in two other tables: the 'table_row_blob' extension table, which stores a Uniform Resource Identifier (URI) for virtual columns of Binary Large Object (BLOB) data type, the 'table_row_clob' extension table, which stores Character Large Object (CLOB) values for virtual columns with TEXT data type. These three types of tables are capable to store all the data type of Big Data including traditional relational data, texts, audios, images, and videos in structured format. (4) The 'table_relationship' extension table allows tenants to create virtual relationships for their virtual tables with any of CTTs or VETs. (5) The 'table_index' extension table is used to add indexes to virtual columns. These indexes reduce the query execution time when tenants retrieve data from VETs. (6) The 'table_primary_key_column' extension table allows tenants to create single or composite virtual primary key for virtual extension columns that are stored in the 'table_column' extension table [5, 8]. This design gives tenants the opportunity of satisfying their different business requirements by choosing from three database models: Multi-tenant Relational Database, Combined Multi-tenant Relational Database and Virtual Relational Database, and Virtual Relational Database [6].

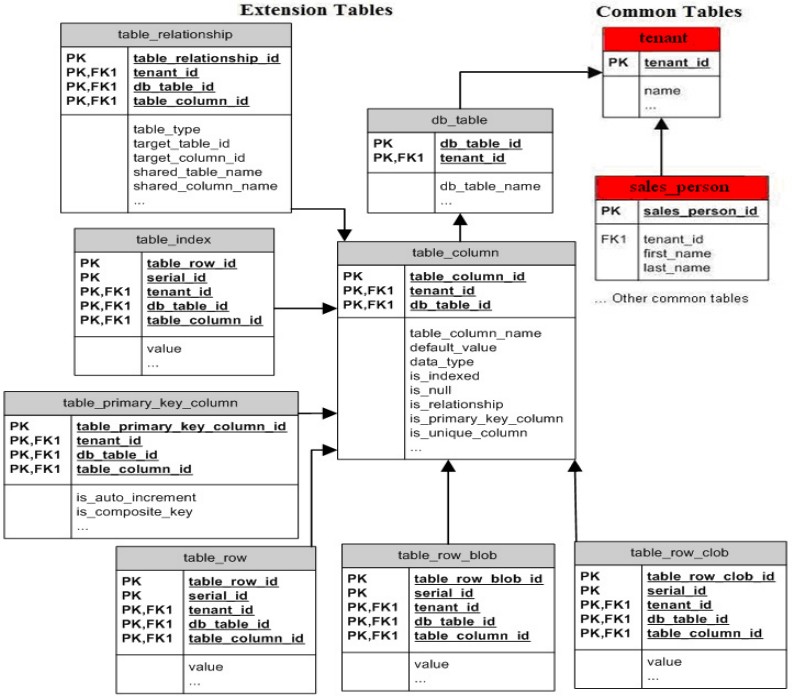

Figure1: Elastic Extension Tables [8]

3 Elastic Extension Tables Schema Handler Service

There are several commercial cloud data management systems (e.g. BigTable, SimpleDB, HyperTable, CouchDB, etc.) that allow end users to manage their data storage using APIs [12]. We apply a similar approach in designing the Elastic Extension Tables Schema Handler Service (EETSHS). This service provides functions that allow tenants to manage their data without having to write SQL queries and backend data management code, by calling data management functions from EET data management APIs which we will cover in our future work.

3.1 Table Management

The EETSHS has three data management functions to manage VET, whereas CTT are managed by both EETSHS and the RDBMS.

- **Create Virtual Tables Function.** This function creates VETs' names for each tenant, and these names are unique names for each tenant. For example, tenant-A can create a VET name 'sales_person', but cannot create the same VET name again in his virtual tables. However, tenant-B can create the 'sales_person' name even if tenant-A already created this virtual table name. This function avoids the redundancy of individual tenant tables, because the 'db_table_name' column of the 'db_table' extension table has UNIQUE constraint.
- **Update Virtual Tables Function.** After creating the table name of the tenant's VET, this name can be updated by calling this function. The updated VET name remains unique for every individual tenant, because of the UNIQUE constraint of the 'db_table_name' column.
- **Delete Virtual Tables Function.** After creating the tenant's VET, the tenant can delete this table by calling this function. Deleting the VET means that the table name and its virtual columns which are stored in the 'table_column' extension table and related to this VET have to be deleted. In addition to the rows, indexes, and constraints that are related to those columns and stored in the other extension tables have also to be deleted. The only case the tenant cannot delete a VET is when it has a master-detail relationship with another VET, and the primary key of the master VET that need to be deleted is foreign key in other details VET.

3.2 Column Management

The EETSHS has three data management functions to manage VETs' columns, whereas CTTs' columns are managed by the RDBMS.

- **Create Virtual Columns Function.** This function creates a virtual column for a VET and specifies its properties by storing the necessary column properties values into the columns of the 'table_column' extension table. The column properties include: (1) the default value of a column that need to be inserted when no data specified for it during creating or updating a virtual row, (2) the data type of the column, (3) index column flag, (4) null column flag, (5) foreign key column flag, (6) primary key column flag, and (7) unique key column flag.
- **Update Virtual Columns Function.** After creating virtual columns for a tenant's VET, the tenant can update any column properties by calling this function.
- **Delete Virtual Columns Function.** After creating virtual columns for a tenant's VET, the tenant can delete any column even if it is a primary key, as long as this column is not a primary key that has foreign keys in any other table pointing to it. This function deletes a column from a VET, and simultaneously deletes the entire rows associate to this column that may be stored in the other extension tables that store rows, relationships, indexes, and primary keys.

3.3 Row Management

The EETSHS has three data management functions to manage CTT's and VET's rows.

- **Create Physical and Virtual Rows Function.** This function creates a tenant table row for a CTT or a VET. The physical rows of CTTs are created in the physical tables of the RDBMS, whereas the virtual rows of VETs are created in the 'table_row', 'table_row_blob', 'table_row_clob', and 'table_index' extension tables. Algorithm 1 presents the details of this function.
- **Update Physical and Virtual Rows Function.** After creating a tenant's table row in a CTT or a VET, the tenant may update this row by calling this function. Algorithm 2 presents the details of this function.
- **Delete Physical and Virtual Rows Function.** After creating a tenant's table row in a CTT or a VET, the tenant may delete this row by calling this function. Algorithm 3 presents the details of this function.

3.4 Relationship Management

The EETSHS has two data management functions to manage virtual relationships between CTTs and VETs.

- **Create Virtual Relationships Function.** This function creates a virtual relationship between CTT and VET, or two VETs. The virtual relationships that we create using this function allows a tenant to choose from any of the three EET database models we mentioned in Section 2. This function stores a master-detail relationship between two tenant's tables in the 'table_relationship' extension table. Simultaneously, it creates in the details VET foreign key columns that refer to the primary key columns of the master CTT or VET.
- **Delete Virtual Relationships Function.** After the tenant creates a virtual master-details relationship between two tables, he can delete this relationship by calling this function. This function deletes the relationship from the 'table_relationship' extension table, deletes from the details VET all the foreign key columns that refer to the primary key columns of the master CTT or VET, and deletes any VET's rows stored in the 'table_row' and 'table_index' extension tables.

In traditional RDBMS, the database administrator cannot update a relationship between two physicals tables. The same case applies for EETSHS; it does not have a function to update virtual relationships. Nevertheless, the tenant can update a relationship by deleting an existing relationship and then creating a new relationship by calling the two functions described in this section.

3.5 Primary Key Management

The EETSHS has two data management functions to manage virtual primary keys of VETs, whereas the primary keys of CTTs are managed by the RDBMS.

- **Create Virtual Primary Keys Function.** This function creates a virtual PRIMARY KEY constraint for a VET column by changing the value of the 'is_primary_key_column' column in the 'table_column' extension table to 'true', and storing the details of the primary key column in the 'table_primary_key_column' extension table. If the column has already data stored in the 'table_row' extension table, then this function copies all of the column's data of the primary key to the 'table_index' extension table. As long as the column is a primary key column then it should be indexed. Otherwise, if the column does not have data then no any data need to be copied to the 'table_index' extension table. This function allows creating single and composite primary keys. In the case when a tenant wants to create a new primary key and the VET has at least one primary key, then this function will store the value 'true' into the 'is_composite_key' column of the 'table_primary_key_column' extension table for the new primary key and the already existing primary keys. Moreover, this function specifies if a primary key is auto incremented, which means

that a unique number is generated when a new row is inserted into a VET. However, this function avoids adding PRIMARY KEY constraint to any column has redundant data.

- **Update Virtual Primary Keys Function.** This function is used for two cases if a column already has a primary key constraint, or if it has not. In the first case, this function deletes the primary key constraint by changing the value of 'is_primary_key_column' in the 'table_column' extension table to 'false', and deletes the details of the primary key column from the 'table_primary_key_column' extension table. If the column has data stored into the 'table_row' extension table, then this function deletes all of column's data of the primary key from the 'table_index' extension table as long as the column is not any more a primary key column then it should not be indexed. Otherwise, if the column does not have data, then no any data need to be deleted from the 'table_index' extension table. Nevertheless, when a tenant deletes a primary key constraint which is part of a composite primary key that consists of two primary keys. Then this function changes the value of the 'is_composite_key' to 'false' for the primary key that will not be deleted from 'table_primary_key_column' extension table. In the second case, when the column is not already a primary key, then this function calls the Create Virtual Primary Key Function. Moreover, this function may update the auto increment property of the primary key either by activating or deactivating it in the 'table_primary_key_column' extension table.

3.6 Index Management

The EETSHS has no specific functions to manage VETs' indexes including primary key, foreign key and custom indexes. However, these indexes are managed in the other functions that discussed in Section 3. The details are listed below:

- **Create Virtual Indexes.** There are four cases to create rows in the 'table_index' extension table: (1) when a tenant creates a virtual master-detail relationship between CTT and VET, or two VETs. In this case, if the master table has table rows, then the primary key of these rows get inserted into the 'table_index' extension table as a foreign keys for the details table. This case occurs in the Create Virtual Relationships Function. (2) When a tenant adds a PRIMARY KEY constraint to a column that already exist in a VET and this column has data, then this data get inserted into 'table_index' extension table. This case occurs in the Create Virtual Primary Keys Function. (3) When a tenant makes a column a custom index column, which is a selective filter in the tenant's query, and this column has data, then this data get inserted into 'table_index' extension table. This case occurs in the Update Virtual Columns Function. (4) Once a VET row inserted in the 'table_row' extension table and this table has indexed columns including primary key, foreign key and custom indexes. Then the values of these indexed columns get inserted into the 'table_index' extension table. This case occurs in the Create Physical and Virtual Rows Function.
- **Update Virtual Indexes.** There is only one case to update virtual indexes in the 'table_index' extension table when the value of a virtual custom index column of a virtual row gets updated in the 'table_row' extension table. Then the same value gets updated in the 'table_index' extension table. This case occurs in the Update Physical and Virtual Rows Function.
- **Delete Virtual Indexes.** There are three cases to delete rows from the 'table_index' extension table: (1) when a tenant deletes a virtual relationship between CTT and VET, or two VETs. In this case, if the master table has data, then the corresponding data get deleted from the table_index' extension table of the details VET. This case occurs in the Delete Physical and Virtual Rows Function. (2) When a tenant updates a custom index column and makes it not indexed column, and this column has data, then this data get deleted from the 'table_index' extension table. This case occurs in the Update Virtual Columns Function. (3) Once a VET row is deleted from the 'table_row' extension table and this row has indexed columns including any of the primary key, foreign key or custom indexes. Then the corresponding index values that related to the deleted row and stored into the 'table_index' extension table get deleted. This case occurs in the Delete Virtual Relationships Function.

4 Sample Algorithms of Elastic Extension Tables Schema Handler Service

In this section, we present three EET data management sample algorithms that are used to allow tenants inserting, updating and deleting rows in CTTs and VETs.

4.1 Creating Physical and Virtual Rows Algorithm

This data management algorithm inserts rows in CTTs and VETs by passing five parameters to it including the tenant ID, table name, table type (CTT or VET), table row matrix, table BLOB matrix, and table CLOB matrix. More details of this algorithm are presented in Algorithm 1.

Definition 1 (Creating Physical and Virtual Rows). T denotes a tenant ID. B denotes a table name. $CPVR_{type}$ denotes the table type whether it is a CTT or a VET. $CPVR_{row}$ denotes a row matrix with 2 rows and n columns. The first row stores a $CPVR_{row\ 0,i}$ that denotes a CTT or VET column name, and the second row stores a $CPVR_{row\ 1,i}$ that denotes a CTT or VET column value. $CPVR_{rowSize}$ denotes the size of $CPVR_{row}$. $CPVR_{blob}$ denotes a BLOB row matrix with 2 rows and n columns. The first row stores a $CPVR_{blob\ 0,j}$ that denotes a CTT or VET BLOB column name, and the second row stores a $CPVR_{blob\ 1,j}$ that denotes a CTT or VET BLOB column value. $CPVR_{blobSize}$ denotes the size of $CPVR_{blob}$. $CPVR_{clob}$ denotes a CLOB row matrix with 2 rows and n columns. The first row stores a $CPVR_{clob\ 0,k}$ that denotes a CTT or VET CLOB column name, and the second row stores a $CPVR_{clob\ 1,k}$ that denotes a CTT or VET CLOB column value. $CPVR_{clobSize}$ denotes the size of $CPVR_{clob}$. $CPVR_{rowID}$ denotes the 'table_row_id' primary key of 'table_row', 'table_row_blob', and 'table_row_clob' extension tables. $CPVR_{serialID}$ denotes the 'serial_id' column in the 'table_row', 'table_row_blob', and 'table_row_clob' extension tables.

Algorithm 1: Creating Physical and Virtual Rows (CPVR)

Input: T, B, $CPVR_{type}$, $CPVR_{row}$, $CPVR_{blob}$, and $CPVR_{clob}$

1. **if** $CPVR_{type}$ = 'CTT' **then**
2. Insert the table row into B in RDBMS for T
3. **else if** $CPVR_{type}$ = 'VET' **then**
4. **if** $CPVR_{row}$ ∉ B **then** /* When the row is not already exist in B */
5. $CPVR_{rowID} \leftarrow$ get max(table_row_id) from **table_row** extension table + 1
6. **for** $i \leftarrow 0$ **to** $CPVR_{rowSize}$ **do**
7. $CPVR_{serialID} \leftarrow i$
8. Insert $CPVR_{rowID}$, $CPVR_{serialID}$, T, B, $CPVR_{row\ 0,i}$, $CPVR_{row\ 1,i}$ into **table_row** extension table
9. **if** $CPVR_{row\ 0,i}$ is indexed column **then**
10. Insert $CPVR_{rowID}$, $CPVR_{serialID}$, T, B, $CPVR_{row\ 0,i}$, $CPVR_{row\ 1,i}$ into **table_index** extension table
11. **end if**
12. $i \leftarrow i + 1$
13. **end for**
14. **for** $j \leftarrow 0$ **to** $CPVR_{blobSize}$ **do**
15. $CPVR_{serialID} \leftarrow j$
16. Insert $CPVR_{rowID}$, $CPVR_{serialID}$, T, B, $CPVR_{blob\ 0,j}$, $CPVR_{blob\ 1,j}$ into **table_row_blob** extension table
17. Store the BLOB file in its designated URI
18. $j \leftarrow j + 1$
19. **end for**
20. **for** $k \leftarrow 0$ **to** $CPVR_{clobSize}$ **do**
21. $CPVR_{serialID} \leftarrow k$
22. Insert $CPVR_{rowID}$, $CPVR_{serialID}$, T, B, $CPVR_{clob\ 0,k}$, $CPVR_{clob\ 1,k}$ into **table_row_clob** extension table
23. $k \leftarrow k + 1$
24. **end for**
25. **end if**
26. **end if**

4.2 Updating Physical and Virtual Rows Algorithm

This data management algorithm updates rows in CTTs and VETs by passing seven parameters to it including the tenant ID, table name, table type, table row matrix, table BLOB matrix, table CLOB matrix, and the table row ID of a VET in the case when a tenant updates a VET row. More details of this algorithm are presented in Algorithm 2.

Definition 2 (Updating Physical and Virtual Rows). T denotes a tenant ID. B denotes a table name. $UPVR_{type}$ denotes the table type whether it is a CTT or a VET. $UPVR_{row}$ denotes a row matrix with 2 rows and n columns. The first row stores a $UPVR_{row\ 0,i}$ that denotes a CTT or VET column name, and the second row stores a $UPVR_{row\ 1,i}$ that denotes a CTT or VET column value. $UPVR_{rowSize}$ denotes the size of $UPVR_{row}$. $UPVR_{blob}$ denotes a row BLOB matrix with 2 rows and n columns. The first row stores a $UPVR_{blob\ 0,j}$ that denotes a CTT or VET BLOB column name, and the second row stores a $UPVR_{blob\ 1,j}$ that denotes a CTT or VET BLOB column value. $UPVR_{blobSize}$denotes the size of $UPVR_{blob}$. $UPVR_{clob}$ denotes a row CLOB matrix with 2 rows and n columns. The first row stores a $UPVR_{clob\ 0,k}$ that denotes a CTT or VET CLOB column name, and the second row stores a $UPVR_{clob\ 1,k}$ that denotes a CTT or VET CLOB column value. $UPVR_{clobSize}$ denotes the size of $UPVR_{clob}$. $UPVR_{rowID}$ denotes the 'table_row_id' primary key of 'table_row', 'table_row_blob', and 'table_row_clob' extension tables. In each virtual table row, this ID is the same row ID for these three extension tables.

Algorithm 2: Updating Physical and Virtual Rows (UPVR)

Input: T, B, $UPVR_{type}$, $UPVR_{row}$, $UPVR_{blob}$, $UPVR_{clob}$, and $UPVR_{rowID}$

1. **if** $UPVR_{type}$ = 'CTT' **then**
2. Update the table row in the **CTT** in RDBMS using T and B query filters
3. **else if** $UPVR_{type}$ = 'VET' **then**
4. **if** $UPVR_{row}$ ∉ B **then** /* When the row is not already exist in B */
5. **for** $i \leftarrow 0$ **to** $UPVR_{rowSize}$ **do**
6. update $UPVR_{row\ 1,i}$ in **table_row** extension table using T, B, $UPVR_{row\ 0,i}$, and $UPVR_{rowID}$ query filters
7. **if** $UPVR_{row\ 0,i}$ is custom index column **then**
8. Update $UPVR_{row\ 1,i}$ in **table_index** extension table using T, B, $UPVR_{row\ 0,i}$, and $UPVR_{rowID}$ query filters
9. **end if**
10. $i \leftarrow i + 1$
11. **end for**
12. **for** $j \leftarrow 0$ **to** $UPVR_{blobSize}$ **do**
13. Update $UPVR_{blob\ 1,j}$ in **table_row_blob** extension table using T, B, $UPVR_{blob\ 0,j}$, and $UPVR_{rowID}$ query filters
14. Delete the existing BLOB file in its designated URI
15. Insert the new BLOB file in its designated URI
16. $j \leftarrow j + 1$
17. **end for**
18. **for** $k \leftarrow 0$ **to** $UPVR_{clobSize}$ **do**
19. Update $UPVR_{clob1,k}$ in **table_row_clob** extension table using T, B, $UPVR_{clob\ 0,k}$, and $UPVR_{rowID}$ query filters
20. $k \leftarrow k + 1$
21. **end for**
22. **end if**
23. **end if**

4.3 Deleting Physical and Virtual Rows Algorithm

This data management algorithm deletes rows from CTTs and VETs by passing five parameters to it including the tenant ID, table name, table type, table row matrix, and the table row ID of a VET in the case when a tenant deletes a VET row. More details of this algorithm are presented in Algorithm 3.

Definition 3 (Deleting Physical and Virtual Rows). T denotes a tenant ID. B denotes a table name. $DPVR_{type}$ denotes the table type whether it is a CTT or a VET. $DPVR_{row}$ denotes a row

matrix with 2 rows and n columns. The first row stores a $DPVR_{row\ 0,i}$ that denotes a CTT or VET column name, and the second row stores a $DPVR_{row\ 1,i}$ that denotes a CTT or VET column value. $DPVR_{rowID}$ denotes the 'table_row_id' primary key of 'table_row', 'table_row_blob', and 'table_row_clob' extension tables. $DPVR_{MDR}$ denotes a master-detail relationship. $DPVR_{detail}$ denotes a details VET of the master table that this algorithm deletes its row. $DPVR_{detailsRow}$ denotes a row in the details table refers to the row that this algorithm aims to delete from the master table. $DPVR_{Relation}$ denotes a list of database relationships that a master CTT may have with details VETs, or a list of relationships that a master VET may have with details CTTs or VETs. $DPVR_{RelationSize}$ denotes the size of $DPVR_{Relation}$. $DPVR_{CTTrelation}$ denotes a list of CTT relationships that a master CTT may have with other details CTTs. $DPVR_{CTTrelationSize}$ denotes the size of $DPVR_{CTTrelation}$. $DPVR_{largeObj}$ denotes a row matrix with 1 row and n columns, each element of this matrix may contains a string of 'BLOB' or 'CLOB'. $DPVR_{PK}$ denotes a CTT or VET list of primary keys.

Algorithm 3: Deleting Physical and Virtual Rows (DPVR)

Input: T, B, $DPVR_{type}$, $DPVR_{row}$, and $DPVR_{rowID}$

1. **if** $DPVR_{type}$ = 'CTT' **then**
2. $DPVR_{PK}$ ← get the primary keys of B from INFORMATION_SCHEMA.TABLE_CONSTRAINTS and INFORMATION_SCHEMA.KEY_COLUMN_USAGE views using T and B query filters
3. **else if** $DPVR_{type}$ = 'VET'
4. $DPVR_{PK}$ ← get the primary keys of B from table_column extension table using T and B query filters
5. **end if**
6. **if** $DPVR_{PK} \neq$ Nil **then** **/* checking if any of the tables that have relationship with B have any row with references to the row that need to be deleted */**
7. **if** $DPVR_{type}$ = 'CTT' **then**
8. $DPVR_{CTTRelation}$ ← get the relationships of B from the INFORMATION_SCHEMA.KEY_COLUMN_USAGE view using T and B query filters
9. **for** $i \leftarrow 0$ **to** $DPVR_{CTTRelationSize}$ **do**
10. **if** $DPVR_{MDR} = DPVR_{CTTRelation\ i} \wedge DPVR_{detailsRow} \in DPVR_{details}$ **then**
11. return /* Exit Algorithm */
12. **end if**
13. $i \leftarrow i + 1$
14. **end for**
15. **end if**
16. $DPVR_{relation}$ ← get the relationships of B from the table_relationship extension table using T and B query filters
17. **for** $j \leftarrow 0$ **to** $DPVR_{relationSize}$ **do**
18. **if** $DPVR_{MDR} = DPVR_{relation\ j} \wedge DPVR_{detailsRow} \in DPVR_{details}$ **then**
19. return /* Exit Algorithm */
20. **end if**
21. $j \leftarrow j + 1$
22. **end for**
23. **end if**
24. **if** $DPVR_{type}$ = 'CTT' **then**
25. Delete the row from the RDBMS using T, B, and $DPVR_{rowID}$ query filters
26. **else if** $DPVR_{type}$ = 'VET' **then**
27. $DPVR_{largeObj}$ ← get from the table_column extension table the BLOB and CLOB objects using T and B query filters
28. **for all** $DPVR_{largeObj}$ **do**
29. **if 'BLOB'** $\in DPVR_{largeObj}$ **then**
30. Delete all BLOB rows from the table_row_blob extension table using T, B, and $DPVR_{rowID}$ query filters
31. Delete the BLOB file from its designated URI
32. **end if**
33. **if 'CLOB'** $\in DPVR_{largeObj}$ **then**
34. Delete all CLOB rows from the table_row_clob extension table using T, B, and $DPVR_{rowID}$ query filters
35. **end if**
36. **end for**
37. Delete rows from table_row extension_table using T, B, and $DPVR_{rowID}$ query filters
38. **end if**

5 Performance Evaluations

We designed the EET multi-tenant database schema in [5, 8], and the EET multi-tenant database architecture to serve multiple tenants in one application instance [7]. However in this paper, the aim of the experiments is evaluating the performance of EETSHS for one tenant. As long as in the multi-tenant database the data of each tenant's user is isolated in a table partition, these experiments can evaluate the effectiveness of managing data for each single tenant from the multi-tenant database. We believe that the multi-tenant database performance needs to be optimized and tested in one single server instance before we consider scale-up and scale-out of multi-tenant databases. We apply this approach to test the effectiveness of running database operations for CTTs and VETs using EETSHS. These experiments compare the performance of the query execution time from a traditional physical table (CTT), and a virtual table (VET) that are stored in the extension tables.

5.1 Experimental Setup

The EETSHS service was implemented in Java 1.6.0, Hibernate 4.0, and Spring 3.1.0. The database is PostgreSQL 8.4 and the application server is Jboss-5.0.0.CR2. Both of database and application server is deployed on the same PC. The operating system is Windows 7 Home Premium, with Intel Core i5 2.40 GHz CPU, 8 GB of RAM memory, and 500 GB of hard disk storage.

5.2 Experimental Data Set

We invoke in our experiment three of the EETSHS functions to insert, update, and delete 1, 10, 50, and 100 rows from the 'product' table. We use this table structure for both the 'product' CTT and VET. There are 200,000 rows stored in these tables that belong to a tenant whose "tenant_id" equals 1000, and the 'db_table_id' of the 'product' VET in the 'db_table' extension table equals 16. The 'product' CTT has a master-detail relationship with 'sales_fact' CTT, whereas the 'product' VET has a master-detail relationship with the 'sales_faact' VET. The db_table_id' of the 'sales_fact' VET in the 'db_table' extension table equals 17. The 'product_id' for both the 'product' CTT and VET equal '300000'. The 'table_row_id' of the 'product' VET equals '50000001'. Figure 2 shows the 'product' and 'sales_fact_tables'. In both the 'product' CTT and VET, we insert in the following columns 'product_id', 'tenant_id', 'product_bus_id', 'standard_cost', 'color', 'price', 'size', and 'weight' the following values respectively 300000, 1000, 123123, 11.5, Red, 100, 10 cm, 140 g. In addition, we update the following columns 'product_bus_id', 'standard_cost', 'color', 'price', 'size', and 'weight' the following values respectively 444333, 12.5, Blue, 105, 105 cm, 155 g. In this data set, we present the values that used in the experiments to test inserting, updating, and deleting one row from the 'product' table. However, we used other values to manage the rest of the rows. In this section, we present the three experiments and the queries of these experiments are shown in Appendix 1.

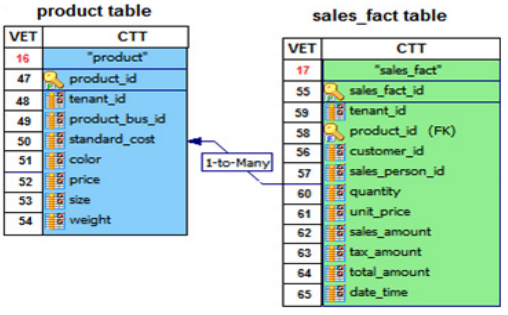

Figure 2: The product and the sales_fact tables structures.

1. **Inserting Physical and Virtual Rows Experiment (Exp. 1).** The aim of this experiment is benchmarking the query execution time of inserting rows into the 'product' CTT and VET. We invoke the Create Physical and Virtual Rows Function from EETSHS which executes Query 1 (Q1) on the 'product' VET. This query comprises of four subsidiary queries, the first query retrieves the maximum number of 'table_row_id' from the 'table_row' extension table. The second query retrieves records from 'table_index' extension table to check if the value of the virtual 'product_id' primary key column name that equals 47 and its value that equals '300000' is already exist or not before inserting the row. The third query inserts eight column values of the 'product' VET in the 'table_row' extension table. The fourth query inserts the values of three column indexes including primary key, foreign key and custom index into the 'table_index' extension table. Whereas the same function executes Query 2 (Q2) on the 'product' CTT to insert the same row values that we insert in Q1.
2. **Updating Physical and Virtual Rows Experiment (Exp. 2).** The aim of this experiment is benchmarking the query execution time of updating rows in the 'product' CTT and VET. We invoke the Update Physical and Virtual Rows Function from EETSHS which executes Query 3 (Q3) on the 'product' VET. This query comprises of three subsidiary queries, the first query retrieves records from 'table_index' extension table to check if the value of the virtual 'product_id' primary key column name that equals 47 and its value that equals '300000' is already exist or not before updating the row. The second query updates six column values of the 'product' VET in the 'table_row' extension table excluding the primary key and foreign key values. The third query updates the custom index value in the 'table_index' extension table. Whereas the same function executes Query 4 (Q4) on the 'product' CTT to update the same row values that we update in Q3.
3. **Deleting Physical and Virtual Rows Experiment (Exp. 3).** The aim of this experiment is benchmarking the query execution time of deleting rows from the 'product' CTT and VET. We invoke the Delete Physical and Virtual Rows Function from EETSHS which executes Query 5 (Q5) on the 'product' VET. This query comprises of five subsidiary queries, the first query retrieves the database relationships that the 'product' VET has with the other VETs and CTTs from the 'table_relationship' extension table. The second query retrieves only the BLOB and CLOB type columns from a VET, and in our experiment the structure of the 'product' VET does not have any of them. In the first query, we found that the 'sales_fact' VET has a master-detail relationship with the 'product' VET. Therefore, the third query checks if the 'sales_fact' VET that is a details table of the master 'product' VET has a row refers to the row that we aim to delete by calling this function. The fourth query deletes all the indexed columns rows that related to the 'product' VET from the 'table_index' extension table. The fifth query, deletes all the columns' rows that related to the 'product' VET from the 'table_row' extension table, and as long as this table does not have BLOB or CLOB columns, then no rows are deleted from the 'table_row_blob' and 'table_row_clob' extension tables. Whereas the same function executes Query 6 (Q6) on the 'product' CTT. This query comprises of five subsidiary queries, the first query retrieves the physical primary keys of the 'product' CTT using a joined query which joins the INFORMATION_SCHEMA.TABLE_CONSTRAINTS and INFORMATION_SCHEMA.KEY_COLUMN_USAGE views. The second query retrieves the details CTTs' names which have a master-detail relationship with the 'product' master CTT from the INFORMATION_SCHEMA.KEY_COLUMN_USAGE view. The third query checks if the 'sales_fact' CTT that is a details table of the master 'product' CTT has a row refers to the row that we aim to delete by calling this function. The fourth query retrieves the details VETs names which have a master-detail relationship with the 'product' CTT. However, since the 'product' CTT does not have any relationship with any VET, therefore no any further queries executed to check if a

details table has row refer to the primary key of the 'product' CTT like what we do in the third query of Q5. The fifth subsidiary query deletes the row from the 'product' CTT.

5.3 Experimental Result

In this section, we show the three experimental results of inserting, updating, and deleting a row from the 'product' CTT or VET as follows:

1. **Inserting Physical and Virtual Rows Experimental Result.** The experimental study of Exp.1 is showing that the execution time of Q1 that performed on the 'Product' VET is approximately 16% slower on average than the execution time of Q2 that perform on the 'product' CTT when we insert 1, 10, 50, and 100 rows. The results of this experiment are shown in Figure 3.
2. **Updating Physical and Virtual Rows Experimental Result.** The experimental study of Exp.2 is showing that the execution time of Q3 that performed on the 'Product' VET is approximately 12% slower on average than the execution time of Q4 that perform on the 'product' CTT when we update 1, 10, 50, and 100 rows. The results of this experiment are shown in Figure 4.
3. **Deleting Physical and Virtual Rows Experimental Result.** The experimental study of Exp.3 is showing that the execution time of Q5 that performed on the 'Product' VET is approximately 73% faster on average than the execution time of Q6 that perform on the 'product' CTT when we delete 1, 10, 50, and 100 rows. The results of this experiment are shown in Figure 5.

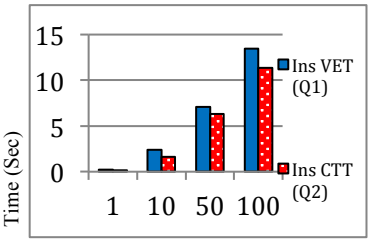

Figure 3: Inserting rows experiment

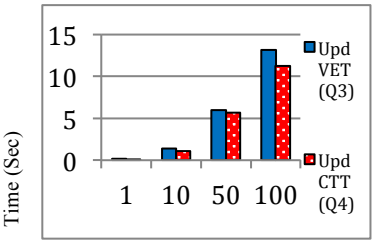

Figure4: Updating rows experiment

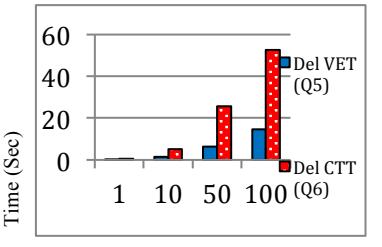

Figure 5: Deleting rows experiment

6 Conclusion

In this paper we propose a multi-tenant data management service based on EET. This service provides functions that allow tenants to manage their data by calling the service functions, without the need of writing SQL queries and backend data management code. Using this service, tenants can create VETs and create VETs' columns, rows, relationships, primary keys, indexes, and other columns' constraints. By using this service tenants can create only CTTs' rows, and database relationships between CTTs and VETs, the rest of the CTT database operations including creating CTTs, CTTs' columns, database relationships between two CTTs, primary keys, indexes, and other columns' constraints can be managed from a traditional RDBMS instead of EETSHS. That is because CTT are shared between multiple tenants, and changing any of these operations affects all the tenants. We present three algorithms that manage CTTs and VETs rows, and we perform several experiments using these algorithms to measure the feasibility and effectiveness of managing data using this service that based on EET. The experimental results show that the query execution time of inserting and updating tenants' CTT rows is slightly faster than for VET rows. This increase in the query execution time of VET is not significant compared to the benefits that this service brings to SaaS and Big Data

applications. The experimental results of deleting tenants' CTT rows are approximately four times slower than VET rows. This increase in the query execution time occurs in CTTs that are the traditional physical tables of EET, because the process of deleting a CTT row is more complicated than VET. As long as EETSHS checks before deleting a CTT row, if the CTT has a master-detail relationship with other CTT or VET, and it checks if any of these tables has any row with references to the row that need to be deleted. In general, these experimental results make this service and EET schema a good candidate for the management of multi-tenant data for software applications in general, and SaaS and Big Data applications in particular. In our future work, we plan to build an administration user interface and APIs for EETSHS that allows tenants' administrators and programmers to manage the tenants' databases.

References

[1] A. V. Hudli, B. Shivaradhya, R. V. Hudli. Level-4 SaaS applications for healthcare industry. In: COMPUTE '09. Bangalore: India; 2009, p. 19.
[2]C. Bezemer, A. Zaidman. Multi-tenant SaaS applications: maintenance dream or nightmare?. In: IWPSE-EVOL '10. Antwerp: Belgium; 2010. p. 88-5.
[3] E. J. Domingo, J. T. Nino, A. L. Lemos, M. L. Lemos, R. C. Palacios, J. M. G. Berbís. CLOUDIO: a cloud computing-oriented multi-tenant architecture for business information systems. In: CLOUD '10. Madrid: Spain; 2010. p.532-2.
[4] F. Chong, G. Carraro, R. Wolter. Multi-tenant data architecture. http://msdn.microsoft.com/en-us/library/aa479086.aspx (Last accessed 9 March, 2014).
[5] H. Yaish, M. Goyal, G. Feuerlicht. An elastic multi-tenant database schema for software as a service. In: DASC '11. Sydney: Australia; 2011. p. 737- 7.
[6] H. Yaish, M. Goyal, G. Feuerlicht. Proxy service for multi-tenant database access. In: ARES '13. Regensburg: Germany; 2013. p.100-18.
[7] H. Yaish, M. Goyal. A multi-tenant database architecture design for software applications. In: BDSE '13. Sydney: Australia; 2013. p. 8.
[8] H. Yaish, M. Goyal, and G. Feuerlicht. Evaluating the performance of multi-tenant elastic extension tables. In: ICCS '14. Carins: Australia; 2014. p. 10.
[9] J. Fiaidhi, I. Bojanova, J. Zhang, L. Zhang. Enforcing multitenancy for cloud computing environments. IT Professional, 14(1) (2012) 16-3.
[10] K. Zhang, X. Zhang, W.Sun, H. Liang, Y. Huang, L. Zeng, X. Liu. A policy-driven approach for software-as-services customization. In: CEC/EEE '07. Tokyo: Japan; 2007. p. 123-8.
[11] L. Guoling. Research on independent SaaS platform. In: ICIME '10. Chengdu: China; 2010. p. 110-4.
[12] S. Sakr, A. Liu, D. M. Batista, M. Alomari. A survey of large scale data management approaches in cloud environments. Communications Surveys & Tutorials, IEEE, 13(3) (2011) 311-26.
[13] T. Kwok, N. Thao, L. Linh. A software as a service with multi-tenancy support for an electronic contract management application In: SCC '08. Hawaii: USA; 2008, p. 179 - 8.

Appendix A. The Experiments Queries.

Q1	**1**	**SELECT** max(table_row_id) From table_row;
	2	**SELECT** * FROM **table_index** WHERE tenant_id=1000 and db_table_id=16 and table_column_id=47 and row_value='300000' order by table_row_id ASC;
	3	**INSERT** into **table_row**(table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,1,1000, '300000',16,47); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,2,1000, '1000',16,48); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,3,1000, '123123',16,49); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,4,1000, '11.5',16,50); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,5,1000, 'Red',16,51); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,6,1000, '100',16,52); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,7,1000, '10 cm',16,53); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,8,1000, '140 g',16,54);
	4	**INSERT** into **table_index** (tenant_id, value, table_row_id, serial_id, db_table_id, table_column_id) values (1000, '300000',50000001,1,16,47); INSERT into table_index (tenant_id, value, table_row_id, serial_id, db_table_id, table_column_id) values (1000, '1000',50000001,2,16,48); INSERT into table_index (tenant_id, value, table_row_id, serial_id, db_table_id, table_column_id) values (1000, '11.5',50000001,4,16,50);
Q2	1	INSERT into product (product_id,tenant_id,product_bus_id,standard_cost,color,price,size,weight) values (300000,1000,'123123',11.5,'Red',100,'10 cm', '140 g');
Q3	**1**	**SELECT** * FROM **table_index** WHERE tenant_id=1000 and db_table_id=16 and table_column_id=47 and row_value='300000' order by table_row_id ASC;
	2	**UPDATE table_row** set value = '444333' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 49 AND table_row_id =50000001; UPDATE table_row set value = '12.5' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 50 AND table_row_id =50000001; UPDATE table_row set value = 'Blue' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 51 AND table_row_id =50000001; UPDATE table_row set value = '105' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 52 AND table_row_id =50000001; UPDATE table_row set value = '105 cm' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 53 AND table_row_id =50000001; UPDATE table_row set value = '155 g' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 54 AND table_row_id =50000001;

	3	**UPDATE table_index** set value = '12.5' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 50 AND table_row_id =50000001;
Q4	1	UPDATE **product** SET product_bus_id = '444333', standard_cost = 12.5, color = 'Blue', price = 105, size = '105 cm', weight = '155 g' WHERE tenant_id = 1000 and product_id = 300000;
Q5	**1**	**SELECT** * FROM **table_relationship** WHERE tenant_id=1000 and (db_table_id=16 or target_table_id=16) order by table_relationship_id;
	2	**SELECT** * FROM **table_column** WHERE data_type= 2 or data_type = 3 and db_table_id=16 order by table_column_id;
	3	**SELECT** * FROM **table_index** WHERE tenant_id=1000 and db_table_id=17 and table_column_id=58 and row_value='300000' order by table_row_id ASC;
	4	**DELETE** from **table_index** WHERE tenant_id = 1000 AND db_table_id = 16 AND table_row_id =50000001;
	5	**DELETE** from **table_row** WHERE tenant_id = 1000 AND db_table_id = 16 AND table_row_id =50000001;
Q6	**1**	**SELECT** c.COLUMN_NAME FROM **INFORMATION_SCHEMA.TABLE_CONSTRAINTS** pk , **INFORMATION_SCHEMA.KEY_COLUMN_USAGE** c where pk.TABLE_NAME = 'product' and CONSTRAINT_TYPE = 'PRIMARY KEY' and c.TABLE_NAME = pk.TABLE_NAME and c.CONSTRAINT_NAME = pk.CONSTRAINT_NAME;
	2	**SELECT** distinct table_name from **INFORMATION_SCHEMA.KEY_COLUMN_USAGE** where column_name in ('product_id');
	3	**SELECT** sales_fact_id FROM **sales_fact** WHERE product_id=300000 limit 1;
	4	**SELECT** * FROM **table_relationship** WHERE tenant_id=1000 and shared_table_name = 'product' order by table_relationship_id;
	5	**DELETE** FROM **product** WHERE tenant_id = 1000 and shr_product_id = 300000;